CS1010S Programming Methodology

Lecture 4

Higher-order Functions

4 Feb 2015

More thinking
Less Coding

Less is more

Don't need to do EVERY Side Quest

Just do ALL the main missions

Post on Forum

(Reflections)

+30 EXP

Tutorials

+200 (attend)

+200 (submit)

Remedial Lessons

Today's Agenda

- Clarifications
- Count Change
  - Recursion
  - Order of Growth
- Higher-order Functions
  - Generalizing Common Patterns
  - Functions as arguments

Watch your syntax

Function call

```
beside(pic1, pic2)

beside(pic)
beside(p1, p2, p3)
```

Conditional

Form 2

```
if expr:
    statements(s)
else:
    statements(s)
```

missing colon

```
if a > 0
print('a > 0!')
else:
    print('a <= 0')
```

no indentation!

What is pass?

Do nothing ☺

Importing Modules

Remember?

```
from runes import *
```

Insight:

Often convenient to have code in different files for code reuse

Importing Modules

- `import X`
  - use `X.name` to refer to objects in X
- `from X import *`
  - creates references to all public objects in X
  - can use plain name
- `from X import a, b, c`
  - creates references to specified objects
  - can now use a and b and c in your program

Counting Change

Problem

Make change for $1, using coins

50¢, 20¢, 10¢, 5¢, 1¢ (assuming unlimited number of coins)

e.g. 50¢ + 50¢

50¢ + 20¢ + 20¢ + 10¢

20¢ + 20 ¢ + 20¢ + 20¢ + 20¢

etc.

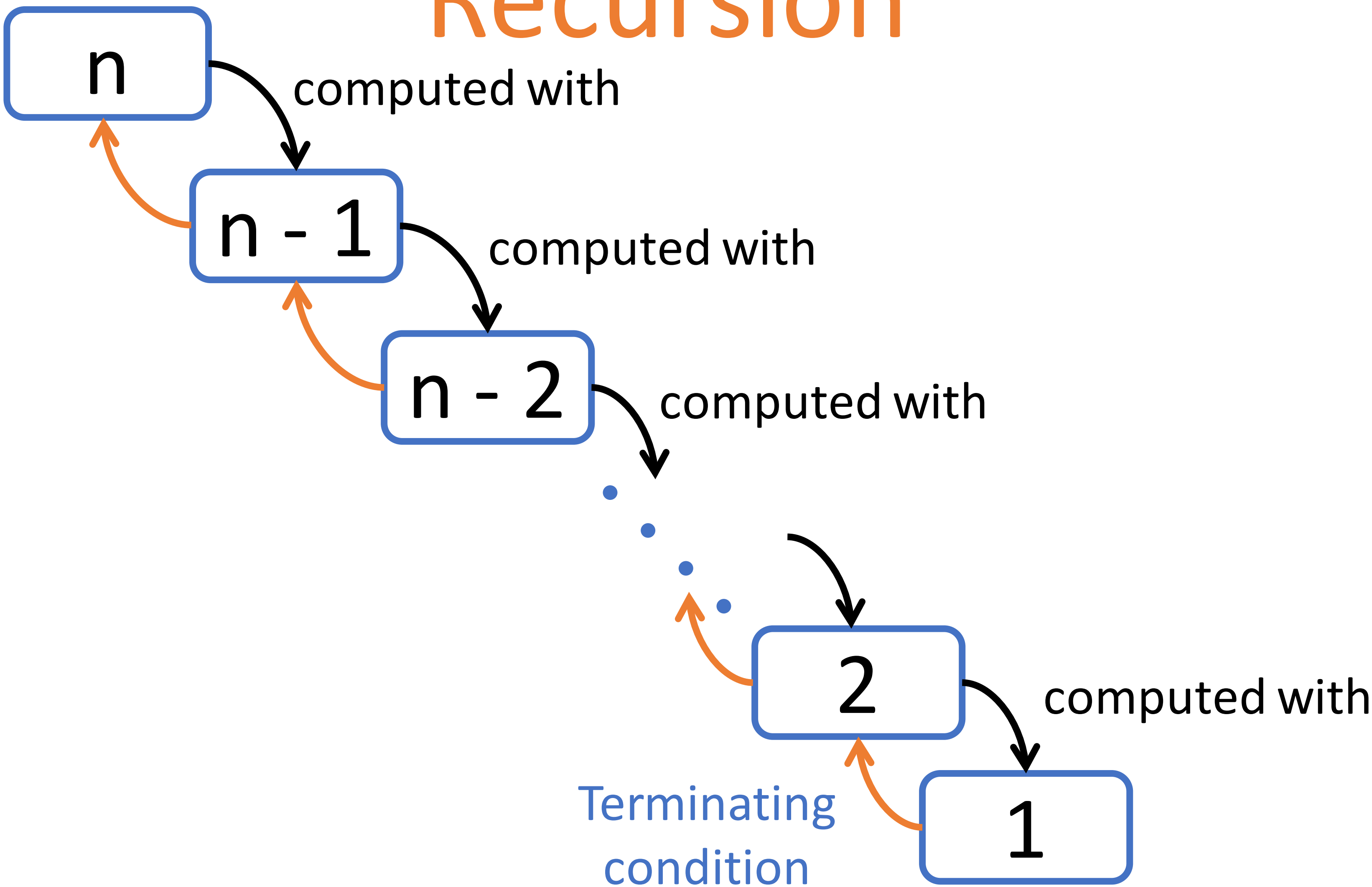
Counting Change

How many ways to do it?

Recap: Recursion

1. Express (divide) the problem into smaller similar problem(s)
2. Solve the problem for a simple (base) case

Recursion


n
computed with
n - 1
computed with
n - 2
computed with
2
computed with
1
Terminating condition

Formulate the problem

- amount: a
  - The amount in cents.
- types-of-coins: $\{d_1, d_2, \dots, d_k\}$
  - e.g. only 50¢ and 20¢

Recursive Idea

Observation: we can divide into two groups

$1

- 50 + 50
- 50 + 20 + 20 + 10

+

- 20 + 20 + 20 + 20 + 20
- 20 + 20 + 20 + 20 + 10 + 10
- etc...

At least one 50 cent coin

No 50 cent coins

Recursive Idea

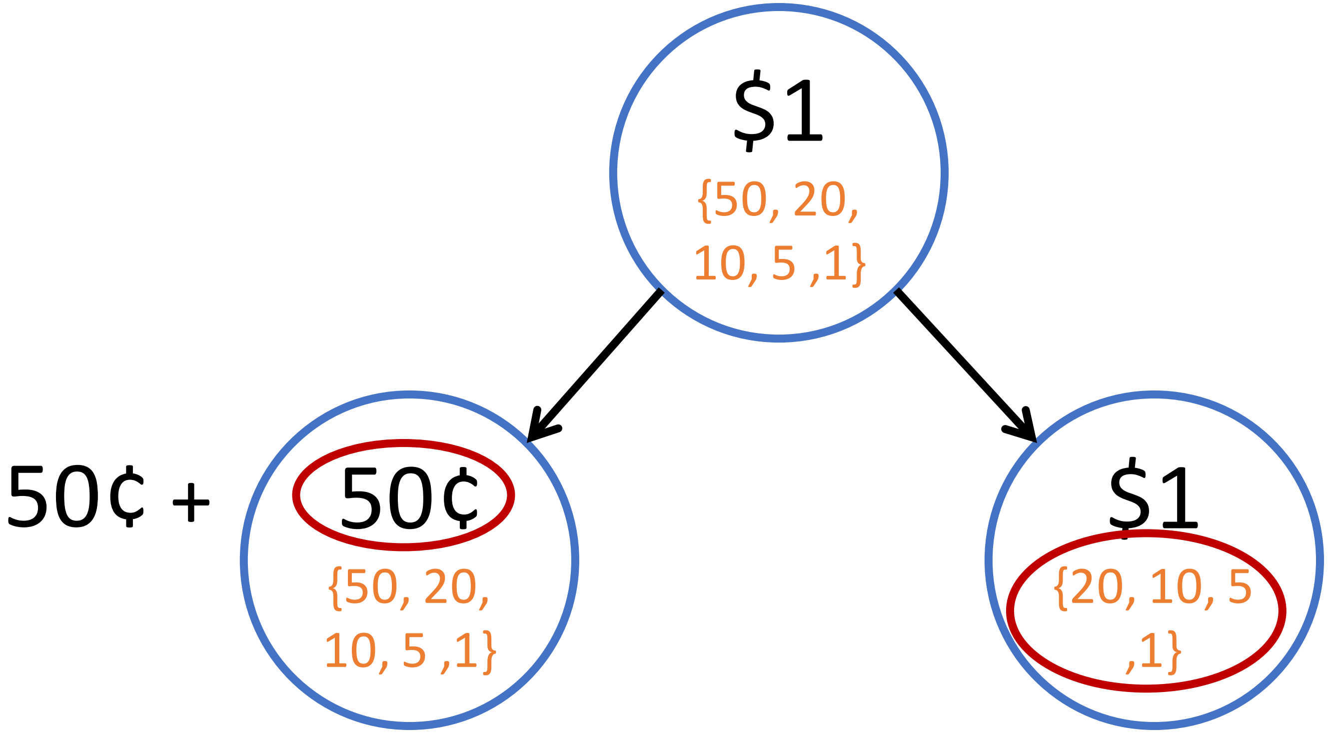
Given a particular set of coins

$$\{d_1, d_2, \ldots, d_n\}$$

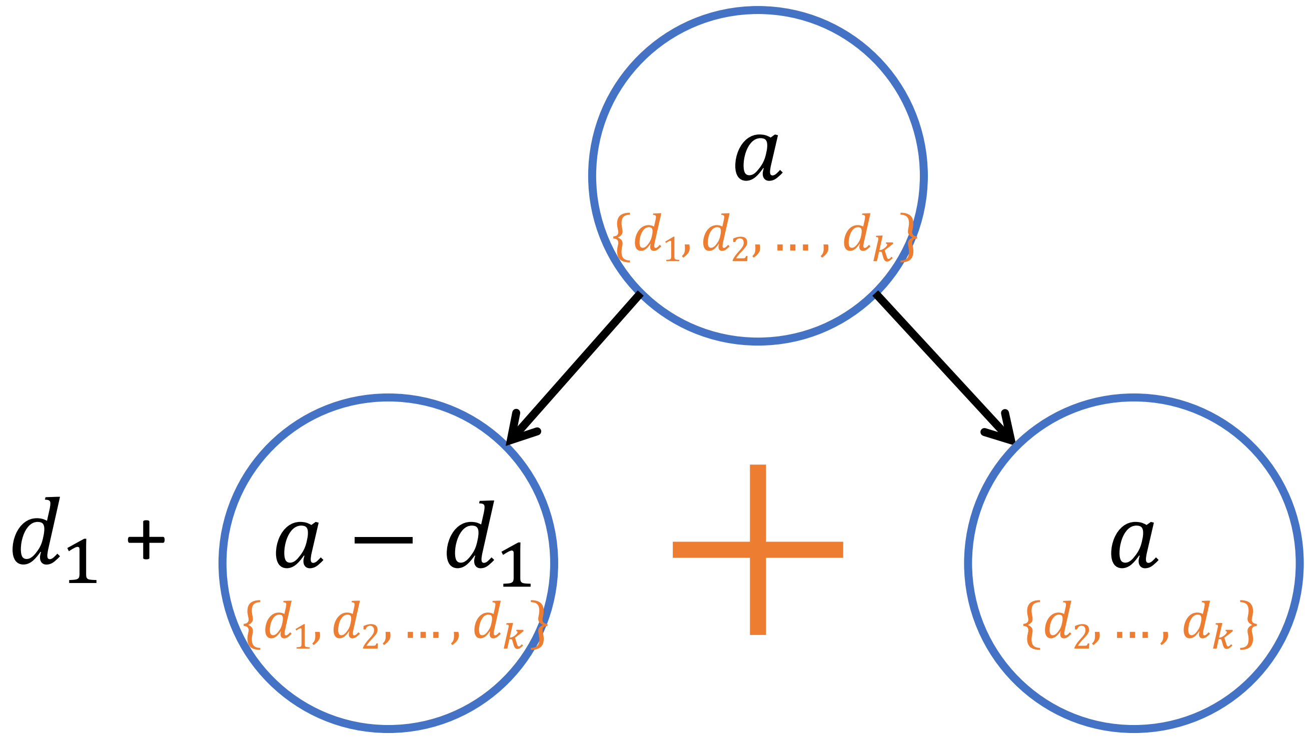
the change for an amount a can be divided into two disjoint and complimentary sets:

1. Those that has least one d_1 coin
2. Those that do not use any d_1 coins

Reduce the problem



In general



Base Cases

- if amount = 0 — only 1 way to make change.
- if amount < 0 — no way to make change, i.e. 0.
- if coins = {} — no way to make change.

Python function

```
def cc(amount, kinds_of_coins):
  if amount == 0:
    return 1
  elif (amount < 0) or (kinds_of_coins == 0):
    return 0
  else:
    return cc(amount – first_denomination(kinds_of_coins),
              kinds_of_coins) +
           cc(amount, kinds_of_coins-1)
```

Using 1 coin for first kind

Without using first kind of coin

```
def first_denomination(kinds_of_coins):
  … <left as an exercise>

def count_change(amount)
  return cc(amount,5)
```

cc(100, 5) → 343

Recursion vs. Iteration

- Counting change is (quite) easily formulated via recursive process.
- Can you write an iterative process to count change?
  - Can, but not easy!

Moral of the story

- In general, an iterative process is (usually) more efficient than a recursive process.
- But sometimes it is hard to devise an iterative solution, whereas a recursive one is straightforward (and more elegant).

Don’t hesitate to write recursive processes. Leave the optimization to the interpreter.

Writing the code is the easy part, figuring out WHAT TO DO is the hard part

Order of Growth

1. Identify the basic computation steps
2. Try a few small values of *n*
3. Extrapolate for really large *n*
4. Look for “worst case” scenario

1. Identify the basic computational step

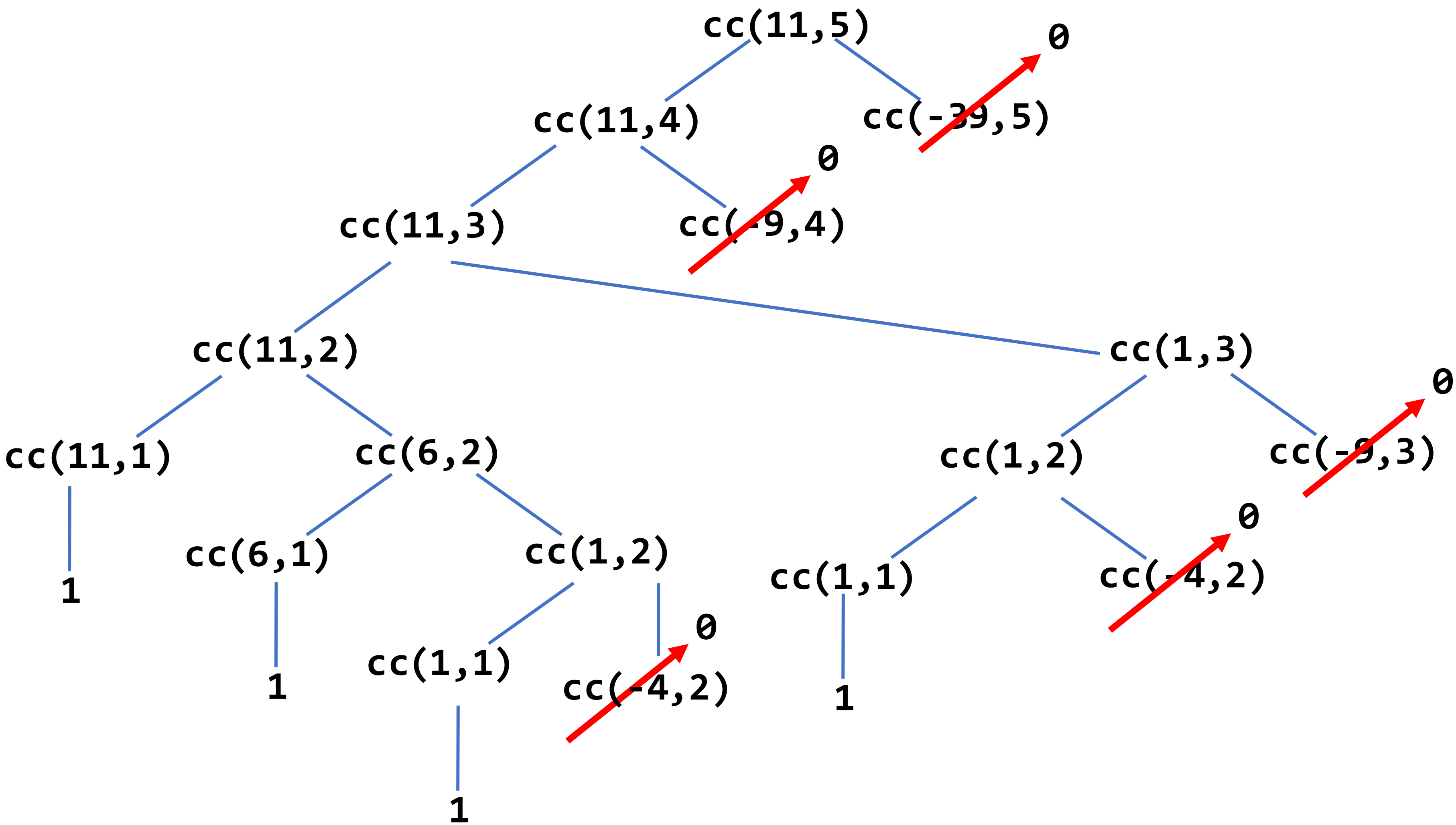
```
def cc(amount, kinds_of_coins):
  if amount == 0:
    return 1
  elif (amount < 0) or (kinds_of_coins == 0):
    return 0
  else:
    return cc(amount – first_denomination(kinds_of_coins),
             kinds_of_coins) +
           cc(amount, kinds_of_coins-1)
```

Using 1 coin for first kind

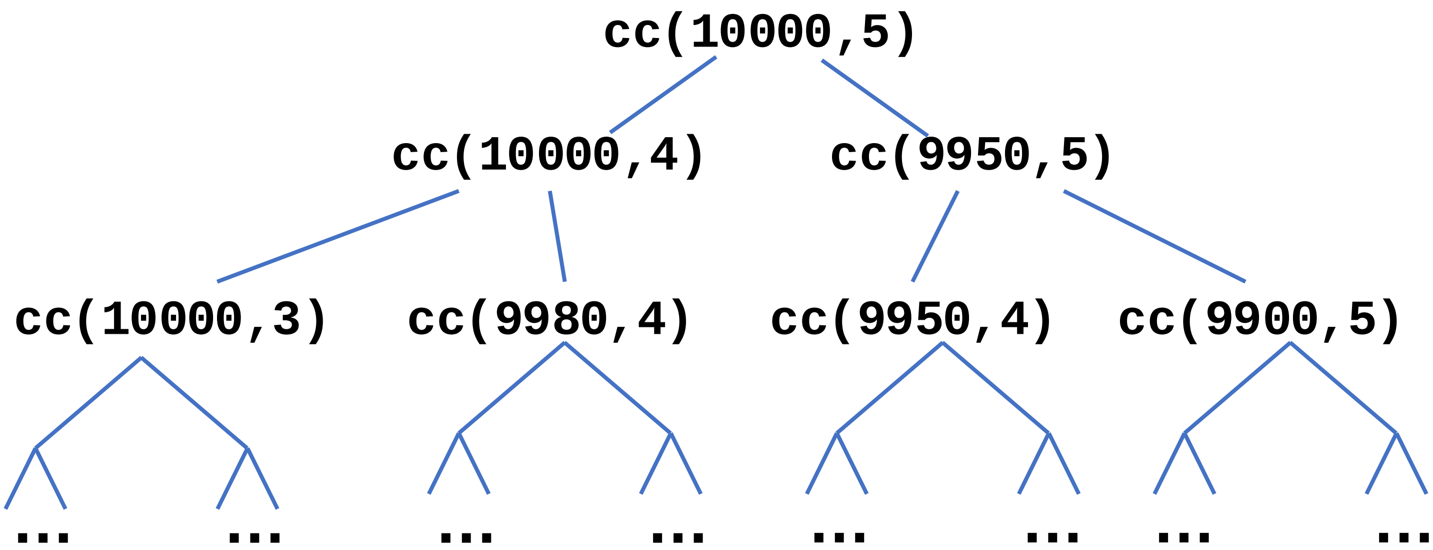
Without using first kind of coin

```
def first_denomination(kinds_of_coins):
  … <left as an exercise>

def count_change(amount)
  return cc(amount,5)
```

2. Try a few small values of n



3. Extrapolate for really large *n*

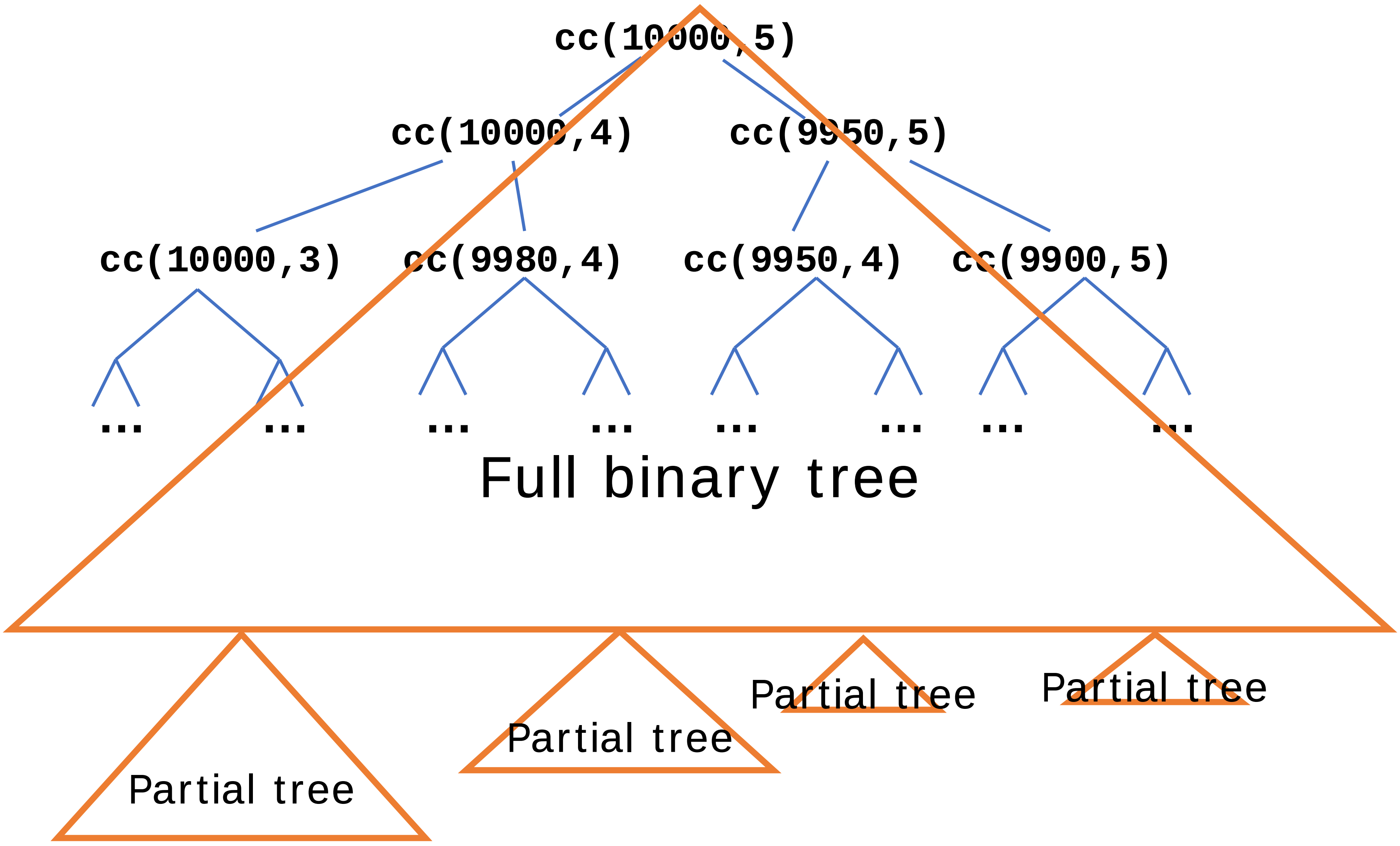


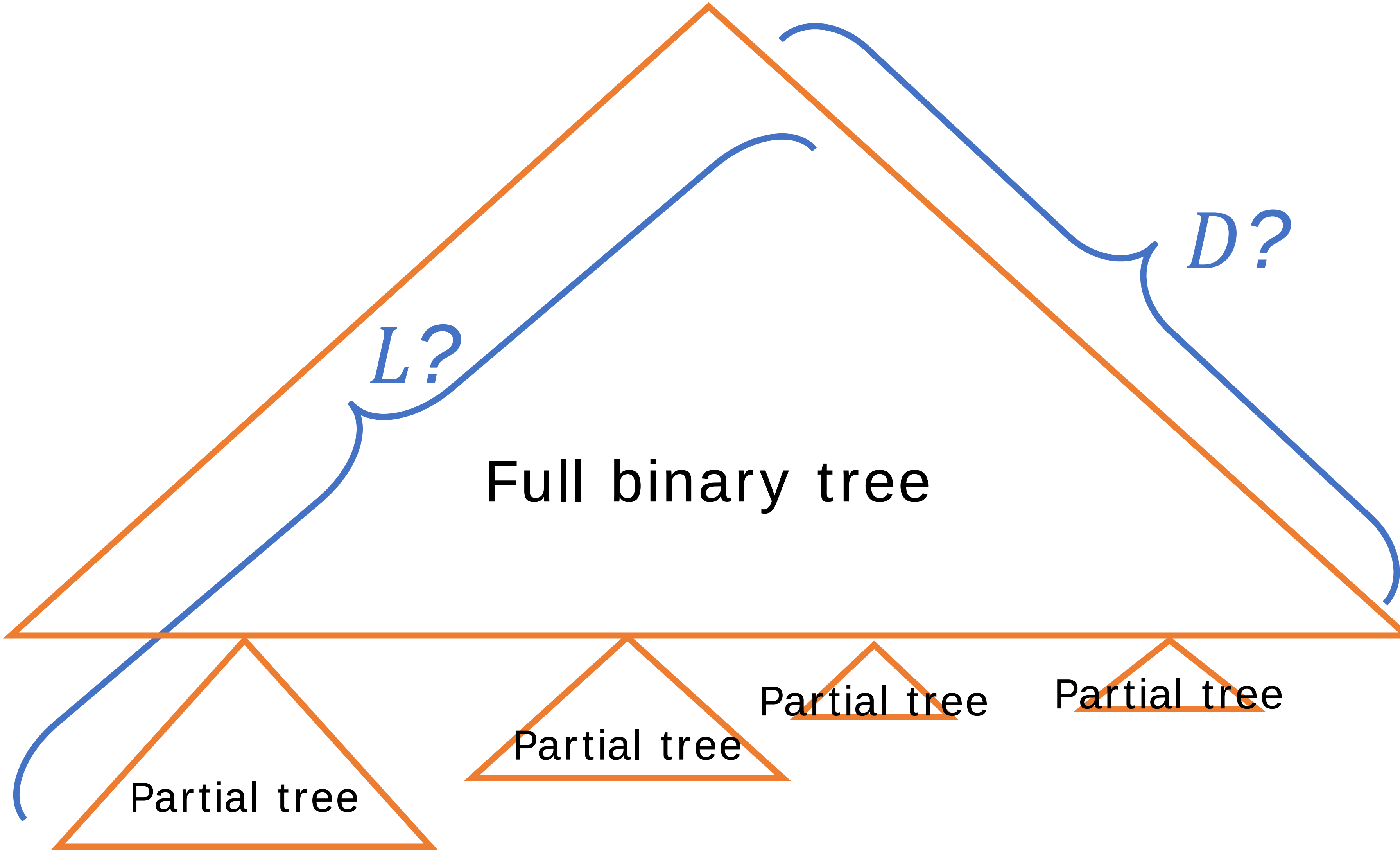
4. Two steps:

(a) Work out the steps in the computation

(b) Generalize to n

4b. Generalize to n





$$D = \frac{a}{\text{largest denomination}}$$

$$L = a + \#\!-\!\text{coin}\!-\!\text{types}$$

Order of Growth

- For large amounts a,
  Time complexity

$$= \text{leaves in the tree}$$
$$= 2^{L}\ (\text{full tree}) - (\text{missing leaves})$$
$$= O\left(2^{L} - \ldots\right)$$
$$= O\left(2^{a+n-\cdots}\right)$$
$$= O(2^{a})$$

Order of Growth in Space

Two main sources:

1. Function Calls (Stack)
   - Look for pending operations & recursive function calls
2. Data Structures (Heap)
   - To be discussed later

Order of Growth

Space complexity

$= \text{depth of entire tree}$

$= L$

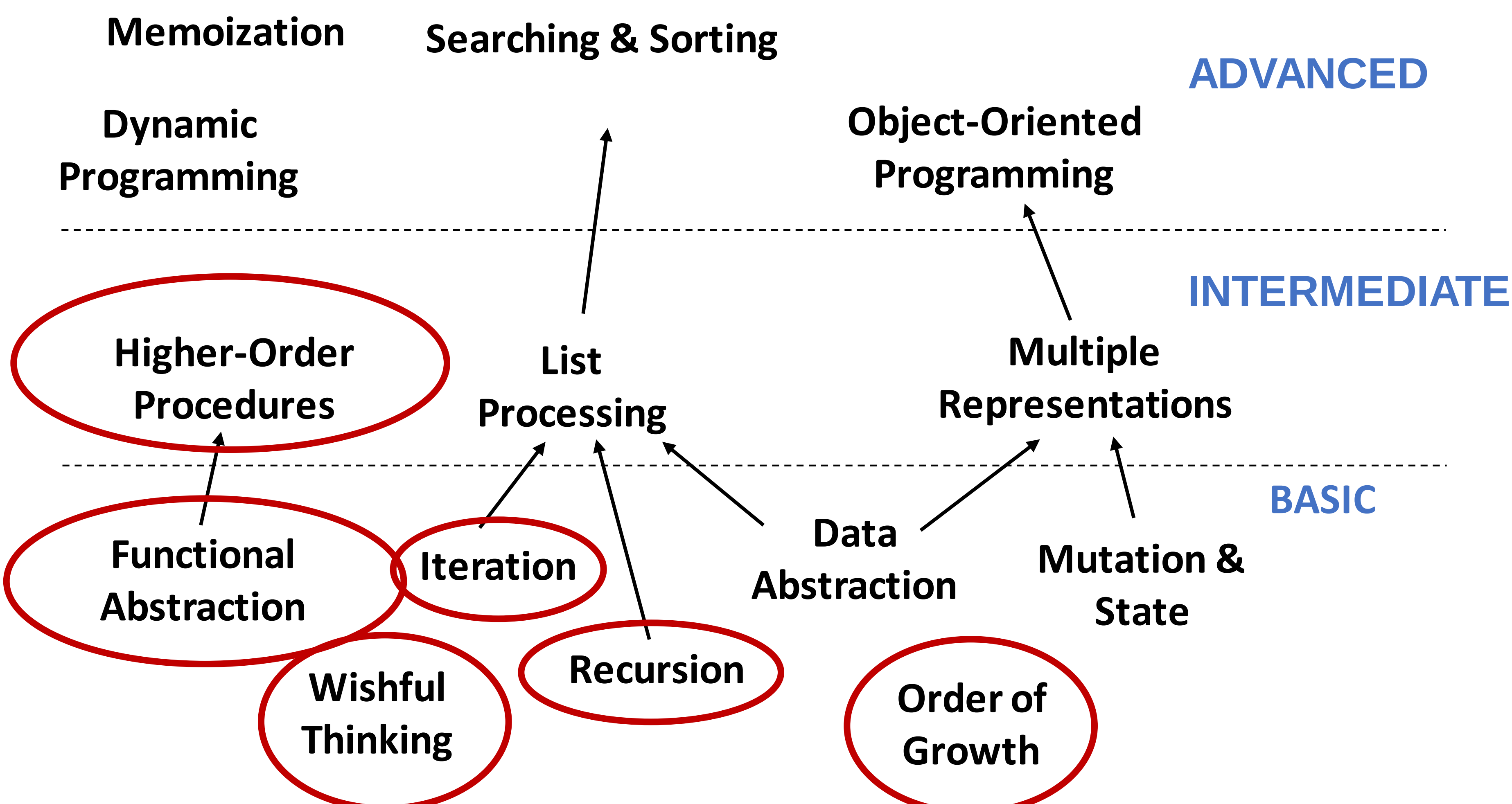
$= a + \#\text{-coin-types}$

$= O(a)$

To Think About

What if you only had a finite number of coins?

CS1010S Road Map



Fundamental **concepts** of computer programming

Abstracting common patterns

Consider the following code to sum all integers in the range *a* to *b*

```
def sum_integers(a,b):
  if a > b:
    return 0
  else:
    return a + sum_integers(a + 1, b)
```

$$\sum_{n=a}^{b} n$$

Abstracting common patterns

Now suppose we want to sum the cubes of numbers in the range *a* to *b*

$$\sum_{n=a}^{b} n^3$$

```
def sum_cubes(a, b):
  if a > b:
    return 0
  else:
    return cube(a) + sum_cubes(a + 1, b)

def cube(n):
  return n * n * n
```

Abstracting common patterns

Finally, we want to sum this series:

a

$$\frac{1}{1\times 3}+\frac{1}{5\times 7}+\frac{1}{9\times 11}+\cdots$$

which converges very slowly to $\pi/_8$

b

```
def pi_sum(a, b):
  if a > b:
    return 0
  else:
    return 1/(a*(a + 2)) + pi_sum(a + 4, b)
```

Abstracting common patterns

- All three functions are very similar.
- Common pattern:

```
def <name>(a, b):
  if a > b:
    return 0
  else:
    return <term>(a) + <name>(<next>(a), b)
```

- Can we abstract this common pattern?

Yes!

```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) +
           sum(term, next(a), next, b)
```

functions

- Note that `term` and `next` are functions.
- Note also that there is a pre-defined function called `sum`. We are over-writing it.

Higher-order functions

- Redefined

```
def sum_cubes(a,b):
  return sum(cube, a,
  inc, b)

def inc(n):
   return n+1
def cube(x):
  return x*x*x
```

```
sum_cubes(1,10) → 3025
```

- Previous

```
def sum_cubes(a, b):
  if a > b:
    return 0
  else:
    return cube(a) +
   sum_cubes(a + 1, b)
```

Higher-order functions

- Redefining `sum_integers`

```
def sum_integers(a, b):
  return sum(identity, a, inc, b)

def identity(x):
   return x


sum_integers(1,10) → 55
```

Higher-order functions

- Alternatively,

```
def sum_integers(a,b):
  return sum(lambda x: x,
             a,
             lambda n: n+1,
             b)
```

anonymous functions

Higher-order functions

- Redefining `pi_sum`

```
def pi_sum(a,b):
  return sum(lambda x: 1/(x*(x+2)),
             a,
             lambda x: x+4,
             b)
```

```
8 * pi_sum(1,1000) → 3.139592655589783
```

Key idea

- `sum` captures a common pattern.
- The other functions (`sum_integers`, `sum_cubes`, `pi_sum`) are special cases of `sum`.

Key idea

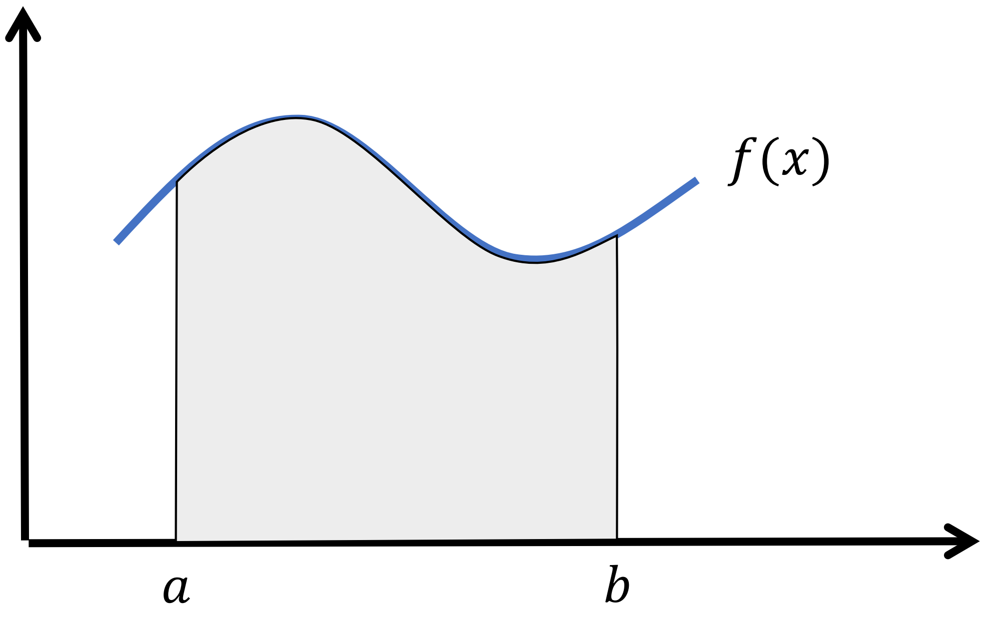
`sum_integers`, `sum_cubes`, `pi_sum` can be defined by providing the appropriate `term` and `next` arguments to `sum`

`sum` is a higher-order function:

takes functions as arguments

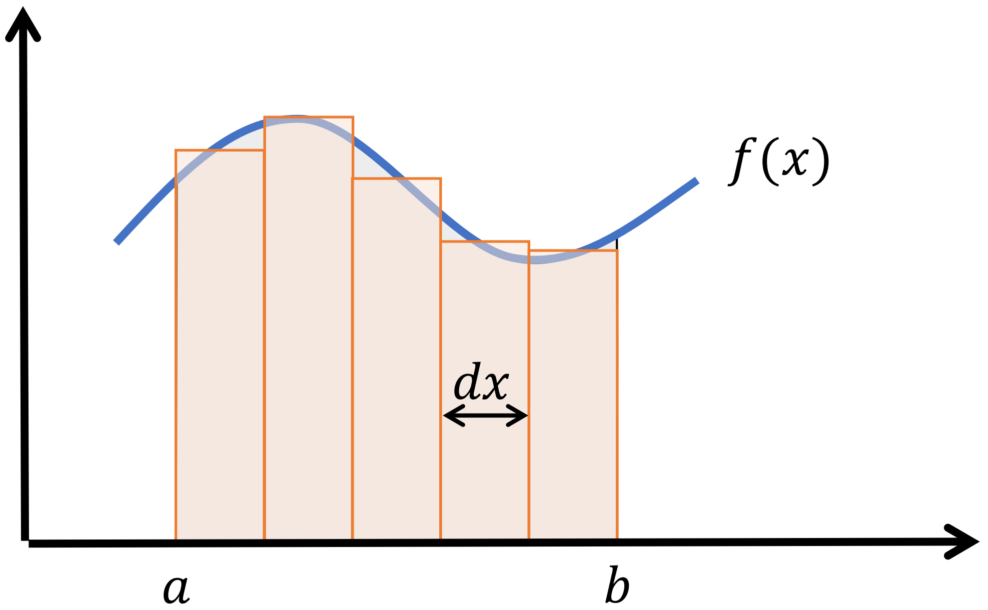
Example: Integration

$$\int_a^b f(x)dx = \text{area under curve}$$



Example: Integration

$$\int_a^b f(x)dx \approx \text{sum area of rectangles}$$



Example: Integration

$$\int_a^b f(x)dx \approx \left(f\left(a + \tfrac{dx}{2}\right) + f\left(a + dx + \tfrac{dx}{2}\right) + f\left(a + 2dx + \tfrac{dx}{2}\right) + \cdots\right) dx$$

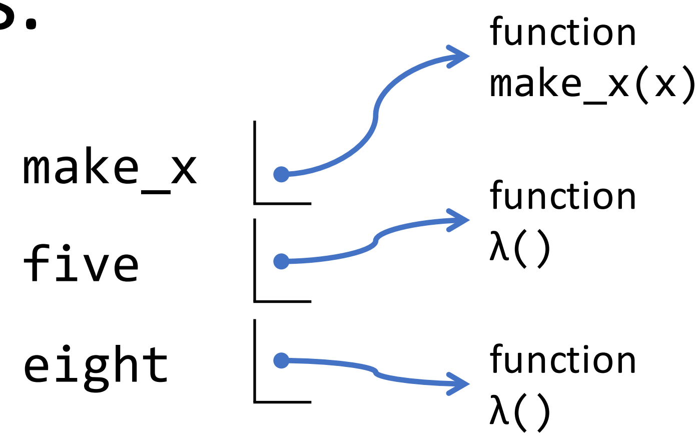
```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) +
           sum(term, next(a), next, b)
```

```
def integral(f, a, b, dx):
    def add_dx(x):
        return x + dx
    return dx * sum(f, a+(dx/2), add_dx, b)

integral(cube, 0, 1, 0.01)
# 0.24998750000000042
# exact value is 1/4
```

Functions as return values

Functions may be returned as values from other functions.

```
def make_x(x):
  return lambda : x
five = make_x(5)
eight = make_x(8)
x = 10
five → <function make.x …>
five() → 5
eight → <function make.x …>
eight() → 8
```



Import to understand what is a function's return type: value or function?

Example: Derivative

- In math, the derivative of $g(x)$ is

$$D(g)(x) = \frac{g(x+dx) - g(x)}{dx}$$

- Derivative transforms a function into another function.

```
def deriv(g):
    dx = 0.00001
    return lambda x: (g(x+dx) - g(x))/dx
```

Derivative

```
from math import sin, pi
cos = deriv(sin)
cos(pi/4) → 0.7071032456451575
cos(pi/2) → -5.000000413701855e-06
    i.e., -5.0000 × 10-6 ≈ 0

cube = lambda x: x*x*x
deriv(cube)(5) → 75.00014999664018
```

Example: Newton's method

To compute root of function $g(x)$, i.e. find x such that $g(x) = 0$

1. Start with initial guess x_0
2. $x \leftarrow x_0$
3. If $g(x) \approx 0$ then stop: answer is x
4. $x \leftarrow x - \frac{g(x)}{D(g)(x)}$
5. Go to step 3

Newton's Method

```python
def newtons_method(g, first_guess):
    dg = deriv(g)
    def improve(x):
        return x - g(x)/dg(x)
    def is_close_enough(v):
        tolerance = 0.0001
        return abs(v) < tolerance

    def attempt(guess):
        if is_close_enough(g(guess)):
            return guess
        else:
            return attempt(improve(guess))

return attempt(first_guess)
```

1. Start with initial guess x_0
2. $x \leftarrow x_0$
3. If $g(x) \approx 0$ then stop: answer is x
4. $x \leftarrow x - \frac{g(x)}{D(g)(x)}$
5. Go to step 3

Computing square root

- Square root of a is the number x such that:

$$x^2 = a$$

- Use Newton's method to solve:

$$g(x) \equiv x^2 - a = 0$$

Newton's method

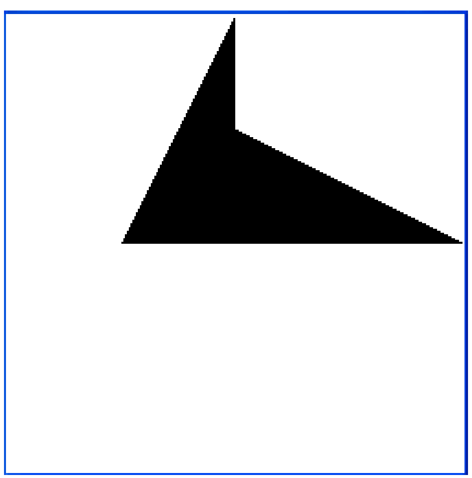
```
square = lambda y: y*y

def sqrt(a):
  return newtons_method(lambda x: square(x)-a,
                                      a/2)
                                     #initial guess is half of a


sqrt(9) → 3.0000153774963274

sqrt(2) → 1.414215695165783
```

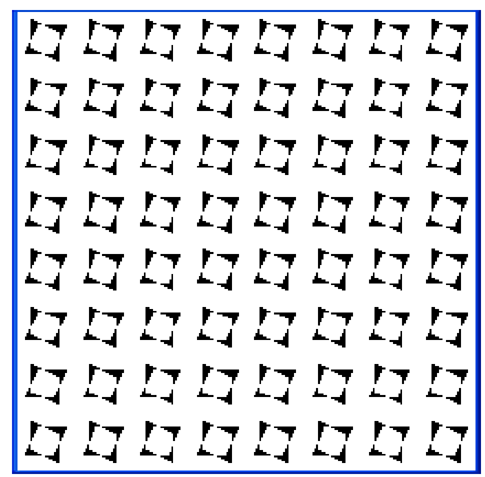
Higher Order Functions Manipulate Other Functions

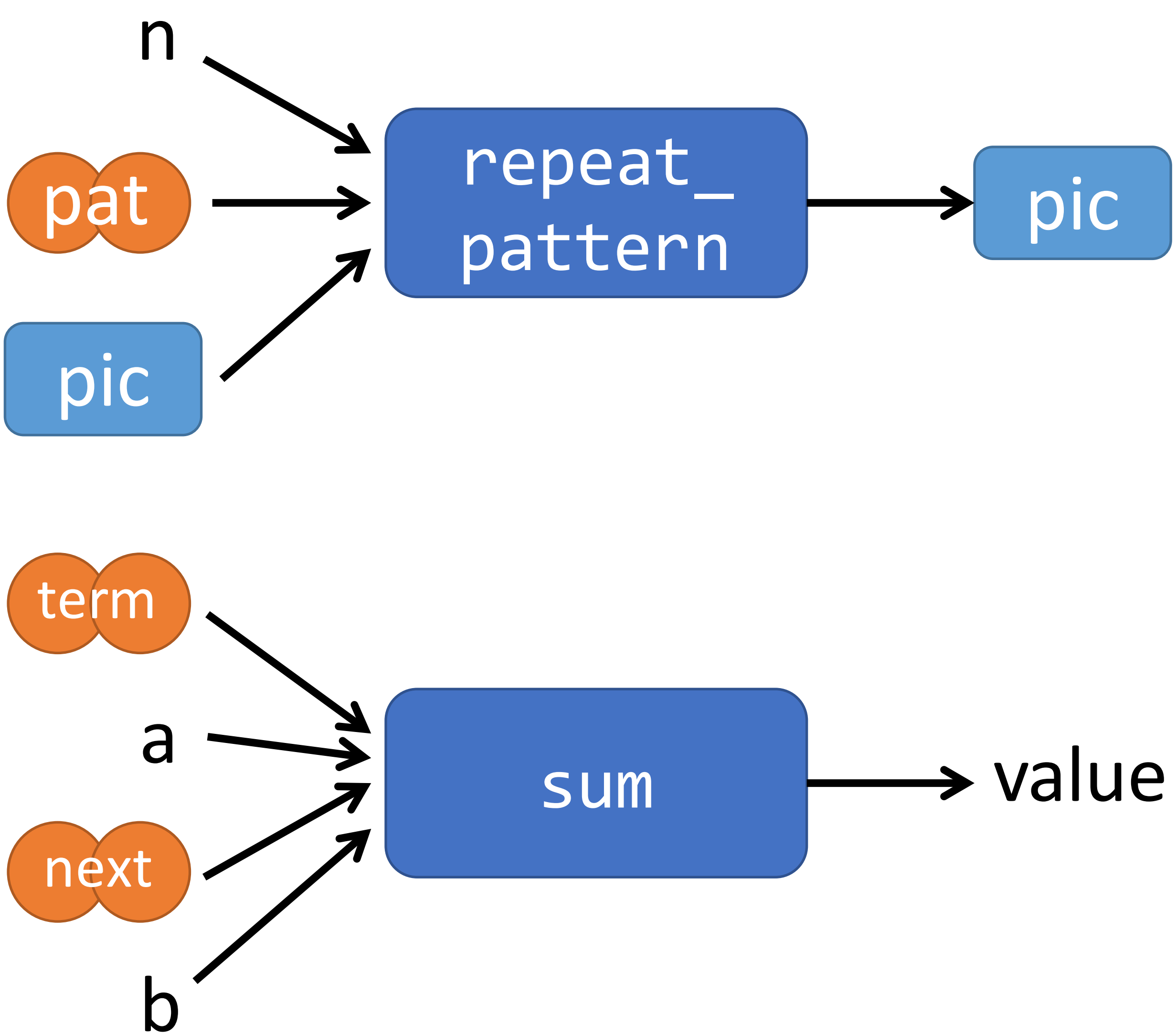
Repeating patterns

```
def repeat_pattern(n, pat, pic):
  if n == 0:
    return pic
  else:
    return pat(repeat_pattern(n-1, pat, pic))
```

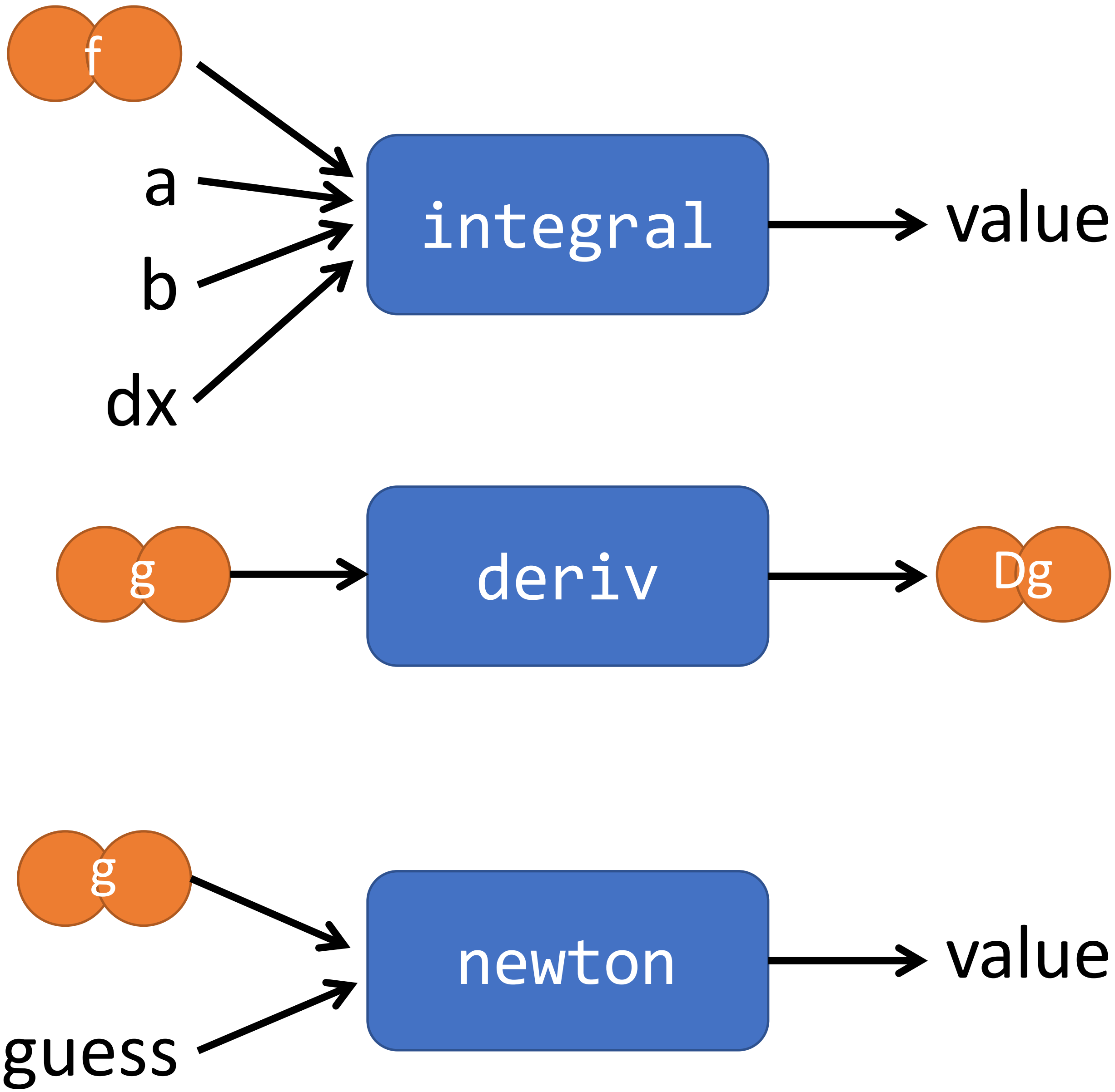
Isn’t this a function also?



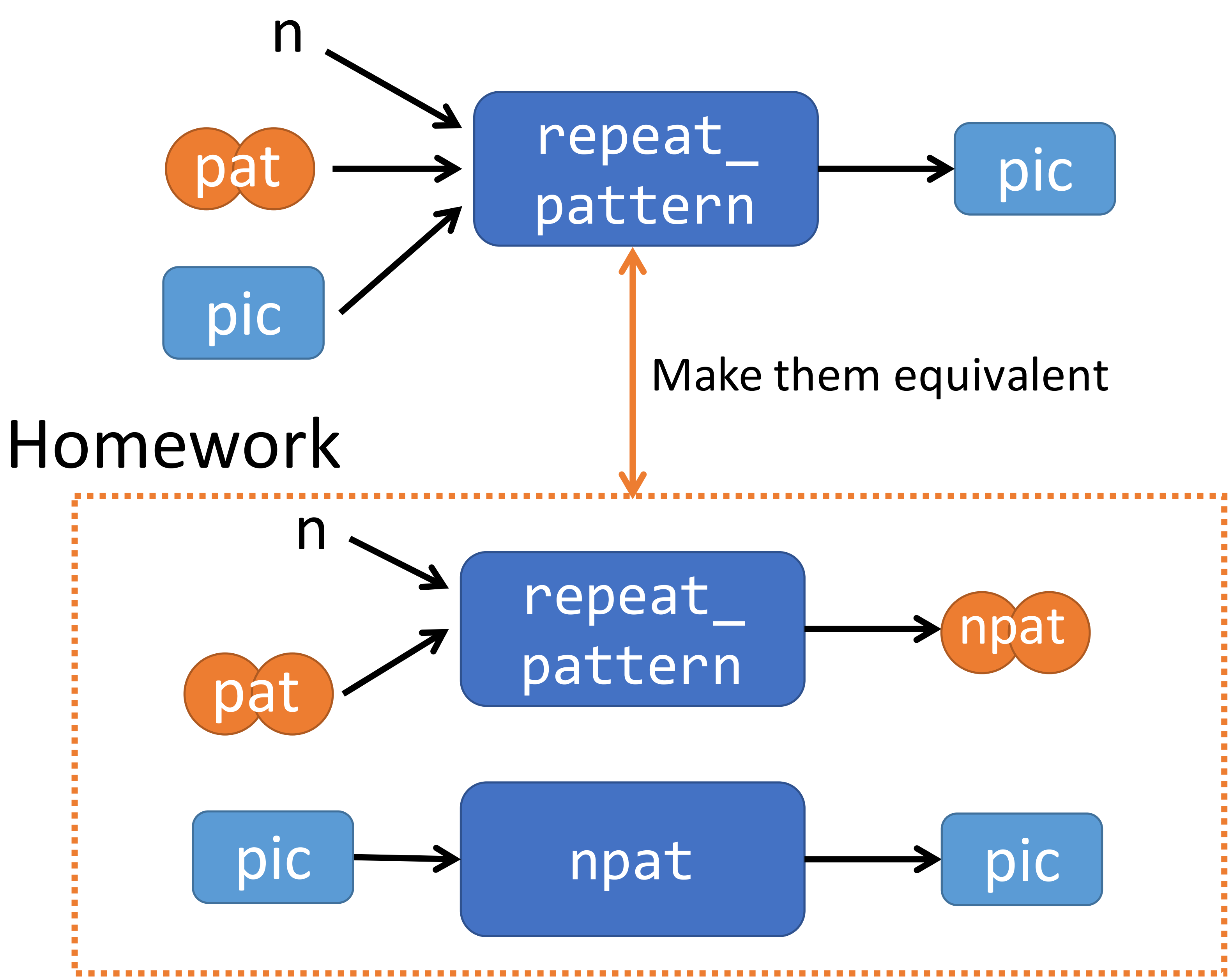




n
pat
pic
repeat_
pattern
pic
term
a
next
b
sum
value



f
a
b
dx
integral
value
g
deriv
Dg
g
guess
newton
value


n
pat
pic
repeat_
pattern
pic
Make them equivalent
Homework
n
pat
repeat_
pattern
npat
pic
npat
pic

Another Example

Compute $\underbrace{f(0) \oplus f(1) \oplus \cdots \oplus f(n)}^{n \text{ binary operations}}$ for some function f, by applying binary operator $\oplus$ n times.

```
def fold(op, f, n):
    if n == 0:
        return f(0);
    else:
        return op(fold(op, f, n-1), f(n))
```

Defining expt with `fold`

Fold: $\overbrace{f(0) \oplus f(1) \oplus \cdots \oplus f(n)}^{n \text{ binary operations}}$ `fold(op, f, n)`

expt: $\overbrace{a \times a \times \cdots \times a}^{n}$

```
f     ⇨    lambda x: a
op    ⇨    lambda x,y: x*y
n     ⇨    n-1
def expt(a, n):
    return fold(lambda x,y: x*y, lambda x: a, n-1)
```

Usage of `fold`

Suppose `sum_of_digits(n)` returns the sum of the digits of *n*. How do we express this as `fold`?

```
def sum_of_digits(n):
  return fold(lambda x,y: x+y,
              lambda k: kth_digit(n,k),
              count_digits(n))
```

Question of the Day:

How do we define `kth_digit` and `count_digits`?

Usage of fold

```
def product_of_digits(n):
  return fold(lambda x,y: x * y,
                   lambda k: kth_digit(n,k),
                count_digits(n))

def sum_of_sqrt_of_digits(n):
    return fold(lambda x,y: x+y,
                   lambda k: sqrt(kth_digit(n,k)),
                   count_digits(n))
```

Recap: Sum of Integers

Consider the following code to sum all integers in the range *a* to *b*

$$\sum_{n=a}^{b} n$$

```
def sum_integers(a,b):
  if a > b:
    return 0
  else:
    return a + sum_integers(a + 1, b)
```

Recap: Sum of Integers

Consider the following code to sum all integers in the range *a* to *b*

$$\sum_{n=a}^{b} n$$

```
def sum_integers(a,b):
    return sum(lambda x: x,
               a,
               lambda n: n+1,
               b)
```

Product of Integers

Consider the following code to sum all integers in the range *a* to *b*

$$\prod_{n=a}^{b} n$$

```
def product_integers(a, b):
    return product(lambda x: x,
                   a,
                   lambda n: n+1,
                   b)
```

What’s this??

Recall: Definition of sum

Definition of sum:

```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) +
              sum(term, next(a), next, b)
```

Definition of product:

```
def product(term, a, next, b):
  if a > b:
    return 1
  else:
    return term(a) *
              product(term, next(a), next, b)
```

A More General Version of fold

```
def fold2(op, term, a, next, b, base):
   if a > b:
     return base
   else:
     return op (term(a),
                fold2(op, term, next(a), next,
                      b, base))
```

abstract as parameters in higher-order function

```
def sum(term, a, next, b):
  return fold2(lambda x,y: x+y, term, a, next, b, 0)

def product(term, a, next, b):
  return fold2(lambda x,y: x*y, term, a, next, b, 1)
```

Please DO NOT memorize the definitions of `fold`, `fold2`, `sum`, `product`, etc.

Don’t Worry about Definitions

1. Functions can be inputs to functions
2. Functions can be returned from functions
3. Both 1 & 2 can happen at the same time!

CS1010S is NOT about memory work.

It is about UNDERSTANDING.

CS1010S is NOT about answers.

It is about PROCESS.

Summary

- Python functions are first-class objects.
  - They may be named by variables.
  - They may be passed as arguments to functions.
  - They may be returned as the results of functions.

Summary

- Higher-order functions capture common programming patterns.
- Functions can be returned as the result of functions

Required Competencies

1. Understand how to use higher-order functions to define specific functions
2. Understand how to define higher-order functions by abstracting patterns

For practice (and to check your understanding.....)

- How would you define `factorial` in terms of `product`?
- How would you define `expt` in terms of `product`?